Chapter 9

Preprocessor Directives

Preprocessor directives are lines in the source file that direct the compiler to alter its normal processing of VAX C source code. Preprocessor directives are not defined formally by the C language, so their implementation may vary from one compiler to another. For example, in most implementations of C running on UNIX systems, the preprocessor is a separate program that operates before the compiler, just as the name preprocessor implies. In VAX C, these directives are executed in an early phase of the compiler.

If you plan to port programs to and from other C implementations, take care in choosing which preprocessor directives to use within your programs. See Section 9.2 for more information about conditional compilation.

The preprocessor directives are introduced by number signs (#) that must appear in column 1 of the source listing. This chapter discusses the following preprocessor directives:

- **#define** and **#undef**—Define macro substitutions and replacements
- **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, **#endif**, and the **defined** operator—Controls under which conditions segments of code are to be compiled or not
- **#include**—Includes source text from an external file
- **#line** and **#**—Specifies a new line number and file name at the terminal, not in the listing file
- **#pragma**—Performs an implementation-specific task

Preprocessor directives are independent of the usual scope rules; they remain in effect from their occurrence until the end of the compilation unit. For more information about the compilation unit, seeChapter 2.

9.1 Macro Definitions (#define and #undef)

The **#define** directive specifies a macro identifier and a token string. The token string is substituted for every subsequent occurrence of that identifier in the program text, unless it occurs inside a **char** constant, a comment, or a quoted string. You use the **#undef** directive to cancel a definition for a macro.

NOTE

Previous versions of this guide refer to these macros as tokens.

The syntax of the **#define** directive is as follows:

```
#define identifier token-string
#define identifier(identifier, ...) token-string
```

If you omit the token string, the identifier is deleted from the text to be processed by the compiler.

After a token string is substituted in the source file, the compiler rescans the source line from the beginning of the substituted text to determine whether the previously inserted text contains identifiers defined by other **#define** directives. If so, the identifiers are replaced by their currently specified token strings. Example 9–1 shows nested **#define** directives.

Example 9–1: Nested Substitution Directives

```
/*  Show multiple substitutions and listing format              */

#define  AUTHOR  james + LAST

main()
{
   int writer,james,michener,joyce;

#define LAST michener
   writer = AUTHOR;

#define LAST joyce
   writer = AUTHOR;
}
```

Compile Example 9–1 with the following command:

```
% vcc -v example.lis -V "show = intermediate" example.c RETURN
```

The following listing results:

```
 1                    /*  Show multiple substitutions and
                         listing format                */
 2
 3                    #define AUTHOR james + LAST
 4
 5                    main()
 6                    {
 7       1            int writer,james,michener,joyce;
 8       1
 9       1            #define LAST michener
10       1            writer = AUTHOR;
                 1    writer = james + LAST;
                 2    writer = james + michener;
11       1
12       1            #define LAST joyce
13       1            writer = AUTHOR;
                 1    writer = james + LAST;
                 2    writer = james + joyce;
14       1            }
```

On the first pass, the compiler replaces the identifier AUTHOR with the token string james + LAST. On the second pass, the compiler replaces the identifier LAST with its currently defined token string value. At line 9, the token string value for LAST is the identifier michener, so michener is substituted at line 10. At line 12, the token string value for LAST is redefined to be the identifier joyce, so joyce is substituted at line 13. The following line is the final text that the compiler processes:

```
writer = james + joyce;
```

You may continue the **#define** directive onto subsequent lines if necessary. You must end each line to be continued with a backslash (\ ). The backslash and newline do not become part of the definition. The first character in the next line is logically adjacent to the character that immediately precedes the backslash. The backslash/newline as a continuation sequence is valid anywhere after the identifier being defined, or anywhere after the left parenthesis in a macro definition.

You can continue comments within the definition line without the backslash /newline. In the following example, all text must appear on the same line unless comments appear in the **white-space**:

#<white-space>define<white-space>identifier[(]

The optional left parenthesis begins a macro parameter list (see Section 9.1.2), and cannot be separated from the identifier.

9.1.1 Constant Identifiers

The first form of the **#define** directive defines a simple substitution, usually a constant for a frequently used identifier. A common use of the directive is to define the end-of-file (EOF) indicator as follows:

```
#define  EOF  (-1)
```

The substitution text for this example is delimited with parentheses to avoid lexical ambiguities when text is substituted in the program. For example:

```
i = EOF;
```

If you substitute the token string –1 for the identifier EOF, then the contiguous characters (=–) may be mistaken for an operator.

9.1.2 Macro Parameters

Some macros include a list of parameters. These macro substitutions look like function calls. If you call a function, control passes from the program to the function object code at run time; if you reference a macro, source code is inserted into the program at compile time. The parameters are replaced by the corresponding arguments and the text is inserted into the program stream. The syntax of a macro definition is as follows:

#define name([parm1[,parm2, ...]]) [token-string]

The name, parm1, parm2, and so forth are identifiers, and token-string is arbitrary text.

After the macro definition, all macro references in the source code with the following form are replaced by the token string from the directive, and any formal parameters that appear in the token string are replaced by the corresponding arguments from the reference. For example, argument arg1 replaces parameter parm1, and so forth, as follows:

name([arg1[,arg2, ...]])

As shown in the syntax of the macro definition, the token string is optional. If you omit the token string from the macro definition, the entire macro reference disappears from the source text.

The token string in the macro definition, as well as actual arguments in a macro reference, may contain other macro references. Substitution occurs, but these nested references are limited to a depth of 64. The maximum number of parameters or arguments is also 64.

The **lowertoupper** macro is a good example of macro substitution. For example:

```
#define  lowertoupper(c)  ((c) - 'a' + 'A')
```

When you reference the **lowertoupper** macro, the compiler replaces the macro keyword and its parameter with the token string from the directive.

Preprocessor directives and the macro references have syntax that is independent of the VAX C language. The following list gives the rules for the specification of macro definitions:

- The macro name and the formal parameters are identifiers that are specified according to the rules for identifiers in the VAX C language.
- You can use spaces, tabs, and comments freely within a **#define** directive. In particular, they can appear anywhere that the delta symbol (Δ) appears in the following example:

  ```
  #Δ defineΔ name(Δ parm1Δ ,Δ parm2Δ )Δ \
  Δ token-stringΔ
  ```

- White space cannot appear between the name and the left parenthesis that introduces the parameter list. White space can appear inside the token string. Also, at least one space, tab, or comment must separate name from **define**. Comments can appear within the token string, but they do not become part of the macro definition.

The following list gives the rules for the specification of macro references:

- Comments and white space characters (spaces, horizontal and vertical tabs, carriage returns, newlines, and form feeds) can be used freely within a macro reference. In particular, they can appear anywhere that the delta symbol appears in the following example:

  ```
  Δ nameΔ (Δ arg1Δ ,Δ arg2Δ )
  ```

- Arguments consist of arbitrary text. Syntactically, they are not restricted to VAX C expressions. They can contain embedded comments and white space. Comments are ignored, but the white space is preserved during the substitution.
- The number of arguments in the reference must match the number of parameters in the macro definition, although individual arguments may be null.
- Commas separate arguments except where they occur inside string or character constants, comments, or parentheses. You must balance parentheses within arguments.

Take care when specifying the token string. Since the token string consists of arbitrary text, replacing parameters with arguments occurs even if a parameter appears inside a character or string constant within the token string. To be recognized, a parameter should be delimited from the surrounding text by white space or punctuation characters, such as parentheses.

You must be careful when specifying macro arguments that use the increment (++), decrement (--), and assignment (such as +=) operators or other arguments that can cause side effects. Function calls are another source of possible side effects. Suppose the **lowertoupper** macro is defined as follows:

```
#define lowertoupper(c) ((c) >= 'a' && (c) <= 'z' ? (c) & 0X5F : (c))
```

Suppose the **lowertoupper** macro is invoked as follows:

```
lowertoupper(p++)
```

When the argument p++ is substituted in the macro definition, the effect within the program stream is as follows:

```
((p++) >= 'a' && (p++) <= 'z' ? (p++) & 0X5F : (p++))
```

The result of this expression may not be what was intended—that is, it may not be the uppercase letter corresponding to the value p++. For this reason, specifying macro arguments that may cause side effects is not good programming practice. Even if you are aware of possible side effects, the token strings within macro definitions are easily changed, which changes the side effects without warning.

9.1.3 Listing Substituted Lines

You can specify optional values in the **vcc** command line to force the listing of all lines that have been modified by macro substitutions. The optional values are **expansion** and **intermediate**. If your **vcc** command line includes **–V show=expansion** (as in the following example), the listing produced by the compiler shows both the original line and the final form of the substituted line. Substituted lines are flagged in the margin with numbers designating the nesting level of substitution.

```
% vcc -v example.lis -V "SHOW=EXPANSION" filepath.c RETURN
```

When you specify the option **intermediate**, the compiler lists all intermediate substitutions with one substitution per line, as in the following command example:

```
% vcc -v example.lis -V"SHOW=INTERMEDIATE" filepath.c RETURN
```

Without one of these two listing options, the compiler only lists the original form of a line.

Example 9–1 shows the effect of the **–V show=intermediate** option. For more information about the format of VAX C compiler listings, seeChapter 2.

9.1.4 Canceling Definitions (#undef)

The following directive cancels a previous definition of the identifier by **#define**:

```
#undef identifier
```

If no previous definition exists, a warning message is generated if you specify **–V standard=portable**.

9.2 Conditional Compilation (#if, #ifdef, #ifndef, #else, #elif, and #endif)

Six directives are available to control conditional compilation. They delimit blocks of statements that are compiled if a certain condition is true. You can nest these directives. The beginning of the block of statements is marked by one of three directives: **#if**, **#ifdef**, or **#ifndef**. Optionally, an alternative block of statements can be set aside with the **#else** or the **#elif** directives. The end of the block is marked by an **#endif** directive.

If the condition checked by **#if**, **#ifdef**, or **#ifndef** is true, VAX C ignores all lines between an **#else** (or **#elif**) and an **#endif** directive.

If the condition is false, the lines between the **#if**, **#ifdef**, or **#ifndef** and an **#else**, (or **#elif**) or **#endif** directive are ignored. The compiler flags ignored lines with the letter X in the compiler listing margin.

The **#if** directive has the following form:

```
#if constant-expression
```

This directive checks whether the constant expression is nonzero (true). The operands must be constants. The increment (++), decrement (--), **sizeof**, pointer (*), address (&), and cast operators are not allowed in the constant expression.

The constant expression in an **#if** directive is subject to text replacement and can contain references to identifiers defined in previous **#define** directives. The replacement occurs before the expression is evaluated.

If an identifier used in the expression is not currently defined, the compiler treats the identifier as though it were the constant 0. A warning message is generated if **-V standard=portable** is specified.

The **#ifdef** directive has the following form:

```
#ifdef identifier
```

This directive checks whether the identifier was previously defined by a **#define** directive.

The **#ifndef** directive has the following form:

```
#ifndef identifier
```

This directive checks to see if the identifier is not defined or if it has been undefined by the **#undef** directive.

The **#else** directive has the following form:

```
#else
```

This directive delimits alternative source lines to be compiled if the condition tested for in the corresponding **#if**, **#ifdef**, or **#ifndef** directive is false. An **#else** directive is optional.

The **#elif** directive has the following form:

```
#elif constant-expression
```

The **#elif** line performs a task similar to the combined use of the **else if** statements in VAX C. This directive delimits alternative source lines to be compiled if the constant expression in the corresponding **#if**, **#ifdef**, or **#ifndef** directive is false and if the additional constant expression presented in the **#elif** line is true. An **#elif** directive is optional.

The **#endif** directive has the following form:

```
#endif
```

This directive ends the scope of the corresponding **#if**, **#ifdef**, or **#ifndef** directives.

9.2.1 The defined Operator

If you need to check to see if many macros are defined, you may want to use the special **defined** operator in a single use of the **#if** line. In this way, you can check for macro definitions in one concise line without having to use many **#ifdef** or **#ifndef** directives.

For example, you might want to check the following macros:

```
#ifdef  token1
printf( "Oh, Mary!\n" )
#endif

#ifndef  token2
printf( "Oh, Mary!\n" )
#endif

#ifdef  token3
printf( "Oh, Mary!\n" )
#endif
```

You can use the **defined** operator in a single use of the **#if** preprocessor directive, as follows:

```
#if  defined (token1)  ||!defined (token2) ||defined (token3)
printf( "Oh, Mary!\n" )
#endif
```

You can use **defined** as you would any other operator. However, you can only use **defined** in the evaluated expression of an **#if** or **#elif** preprocessor directive.

9.3 File Inclusion (#include)

The **#include** directive inserts external text into the macro stream delivered to the compiler. Often, global definitions for use with the system library interfaces are included in the program stream with the **#include** directive. The **#include** directives may be nested to a depth determined by the limit of the number of concurrent open files for the process. The VAX C compiler imposes no inherent limitation on the nesting level of inclusion.

The following sections describe the forms of the **#include** directive.

9.3.1 Inclusion Using Angle Brackets (<>)

The first form of the directive is as follows:

#include <file-path>

The identifier file-path must be a valid file path name. The compiler first searches for the file relative to any directories specified with the **–I** option on the **vcc** command line. The compiler searches the directories in the order that they are specified on the command line. If the file is not found in any of these directories, the compiler looks for the file in the directory /usr/include. If the file is found, it is included in the compilation. If it is not found, the compiler generates an error.

9.3.2 Inclusion Using Quotation Marks (" ")

The second form of the **#include** preprocessor directive is as follows:

#include "file-path"

The identifier file-path must be a valid file path name. The compiler first searches for the file relative to the directory in which the including source file was found. If it is not found there, the compiler next searches for the file relative to any directories specified with the **–I** option on the **vcc** command. The compiler searches the directories in the order that they are specified on the command line. If the file is not found in any of these directories, the compiler looks for the file in the directory /usr/include. If the file is found, it is included in the compilation. If it is not found, the compiler generates an error.

9.3.3 Macro Substitution in #include Directives

VAX C allows macro substitution within the **#include** preprocessor directive.

For instance, if you want to include a file name, you can combine the **#define** and **#include** directives, as shown by the following example:

```
#define  token1  "file.ext"
#include token1
```

If you use defined macros in **#include** directives, the macros must evaluate to one of the two following acceptable **#include** file specifications or the use generates an error message:

<file-spec>
"file-spec"

9.4 Specifying Line Numbers (#line and #)

The VAX C compiler keeps track of information about relative line numbers in each file involved in the compilation and uses the number when it delivers diagnostic messages to the terminal. The compiler increments the subsequent lines from the line number specified by the **#line** directive. The directive can also specify a new file specification for the program source file. The **#line** directive does not change the line numbers in your compilation listing, only the line numbers given in messages (for example, error messages) sent to the terminal screen. This directive is useful for locating errors in text that is included using the **#include** preprocessor directive.

The formats of the **#line** directive are as follows:

#line constant identifier
#line constant string
constant identifier
constant string

The compiler gives the line following a **#line** directive the number specified by the parameter constant. You can specify the second parameter as either a VAX C identifier or a character-string constant. It supplies the valid file names. The character string must not exceed 255 characters.

9.5 Implementation-Specific Preprocessor Directive (#pragma)

This section describes the implementation-specific preprocessor directives, or pragmas, that are available in the VAX C compiler. The **#pragma** directive is a standard method for implementing features that vary from one compiler to the next.

Note that **#pragma** directives are subject to macro expansion. A macro reference can occur anywhere after the keyword **pragma**. The following example demonstrates this feature using the **#pragma inline** directive:

```
#define opt inline
#define f func
#pragma opt(f)
```

The **#pragma** directive becomes #pragma inline (func) after both macros are expanded.

The following sections describe the **#pragma** directives.

9.5.1 #pragma [no]builtins Directive

The **#pragma [no]builtins** directive disables or provides access to the VAX C predefined functions. These functions do not result in a reference to a function in the run-time library or in your program. Instead, the compiler generates the machine instructions necessary to carry out the function directly at the call site. (For information on available built-in functions, see Chapter 10.)

The **#pragma [no]builtins** directive has the following format:

```
#pragma builtins
#pragma nobuiltins
```

9.5.2 #pragma [no]inline Directive

The preprocessor directive **#pragma inline** suggests to the compiler that it provide inline expansion of the specified functions. Inline expansion of functions reduces execution time by replacing the function call with code that performs the actions of the original function code.

By default, VAX C attempts to provide inline expansion for all functions. The compiler also uses the following function characteristics to determine if it can provide inline expansion:

- Size
- Number of times the function is called
- Absence of the restrictions described in Section 9.5.2.1

The **#pragma inline** directive requests that the compiler attempt to provide inline code regardless of the size or number of times the specified functions are called. Functions that contain one of the restrictions described in Section 9.5.2.1 are never expanded inline, regardless of the use of the **#pragma inline**.

The **#pragma inline** directive has the following format:

```
#pragma inline (id, . . . )
```

id
Is a C function identifier.

For instance, the following example specifies that the functions push and pop be expanded inline throughout the module in which the **#pragma inline** appears:

```
void push( int );
int pop(void);

#pragma inline( push, pop)

int stack[100];
int *stackp = &stack;

void push(int x)
{
    if (stackp == &stack)
        *stackp = x;
    else
        *stackp++ = x;
}

int pop()
{
    return *stackp--;
}

main()
{
  push(1);
  printf("The top of stack is now %d \n",pop());
}
```

The **-V**"OPTIMIZE=NOINLINE" and the **-V**"NOOPTIMIZE" options disable all **#pragma inline** directives that appear in your source code.

The **#pragma noline** can be used selectively to identify functions that are not to be expanded inline, even when the **-V**"OPTIMIZE=INLINE" option is used on the **vcc** command line. The **#pragma noinline** directive has the following format:

#pragma noinline (id, . . .)

id
Is a C function identifier.

9.5.2.1 Restrictions on Inline Expansion

If a function is to be expanded inline, you must place the function definition in the same module as the function call. The definition can appear either before or after the function call.

Functions cannot be expanded inline if they perform the following tasks:

- Take the address of an argument.
- Use an index expression that is not a compile-time constant in an array that is a field of a **struct** argument. An argument that is a pointer to a **struct** is not restricted.
- Use the *varargs* package to access the function's arguments because they require arguments to be in adjacent memory locations, and inline expansion may violate that requirement.

When automatic inline expansion is not possible, no error or warning message is produced. When you explicitly request inline expansion by using the **#pragma inline** directive, a warning message is produced if inline expansion cannot be done.

9.5.3 #pragma [no]member_alignment Directive

By default, VAX C/ULTRIX aligns structure members on their natural boundaries. However, you can use **#pragma nomember_alignment** to explicitly specify member alignment on byte boundaries.

The **#pragma member_alignment** directive has the following format:

```
#pragma [no]member_alignment
```

When **#pragma member_alignment** is used (or defaulted), the compiler aligns structure members on the next boundary appropriate to the type of the member, rather than on the next byte. For instance, a **long** variable is aligned on the next longword boundary; a **short** variable is aligned on the next word boundary.

Consider the following example:

```
#pragma nomember_alignment

struct x {
           char c;
           int b;
           };

#pragma member_alignment

struct y {
          char c;       /*3 bytes of filler follow c */
          int b;
          };

main ()

{
        printf( "The sizeof y is: %d\n", sizeof (struct y) );
        printf( "The sizeof x is: %d\n", sizeof (struct x) );
}
```

When this example is executed, it shows the difference between **#pragma member_alignment** and the directive **#pragma nomember_alignment**. The difference can also be seen by compiling **–V** x.lis **–V show=symbols** and comparing the listed information for the two structures.

Once used, the **nomember_alignment** pragma remains in effect until the **member_alignment** pragma is encountered.

9.5.4 #pragma [no]standard Directive

Use **#pragma nostandard** to tell VAX C to ignore the current setting of the **–V standard=portable** option until further notice. It has no effect if the qualifier is not specified.

The **#pragma nostandard** directive has the following format:

```
#pragma nostandard
```

Use **#pragma standard** to tell VAX C to reinstate the setting of the **–V standard=portable** option. This pragma does not turn on portability checking if the **–V standard=portable** option is not specified on the **vcc** command line.

The **#pragma standard** directive has the following format:

```
#pragma standard
```

The **nostandard** and **standard** pragmas are together to define regions of source code where portability diagnostics are never to be issued. The following example demonstrates the use of these pragmas:

```
#pragma nostandard
extern noshare FILE *stdin, *stdout, *stderr;
#pragma standard
```

In this example, **nostandard** prevents the NONPORTCLASS diagnostic from being issued against the **noshare** storage-class modifier, which is VAX C specific.